Strong and Weak Scaling of Hybrid Parallelism for Volume Rendering on Large, Multi-core Systems

Mark Howison, E. Wes Bethel, Hank Childs
Computational Research Division
Lawrence Berkeley National Laboratory
One Cyclotron Road, Berkeley, CA 94720, USA
Email: {mhowison,ewbethel,hchilds}@lbl.gov

***Abstract*—With the computing industry trending towards multi-core processors, we study how a standard visualization algorithm, ray-casting volume rendering, can benefit from a hybrid parallelism approach. Hybrid parallelism provides the best of both worlds: using distributed-memory parallelism on a large numbers of cores increases available FLOPs and memory, and exploiting the shared memory parallelism available on each node ensures that each node performs its portion of the larger calculation as efficiently as possible. We demonstrate results from weak and strong scaling studies, at levels of concurrency ranging from 1,728 to 216,000, and with datasets as large as 12.2 trillion cells. The greatest benefit from hybrid parallelism lies in the communication portion of the algorithm, the dominant cost at higher levels of concurrency. We show that reducing the number of participants with a hybrid approach significantly improves performance.**

I. Introduction

It is well accepted that the future of parallel computing involves chips that are comprised of many (smaller) cores. With this trend towards more cores on a chip, many in the HPC community have expressed concern that parallel programming languages, models, and execution frameworks that have worked well to-date on single-core massively parallel systems may "face diminishing returns" as the number of computing cores on a chip increase [1]. In the broader high-performance computing community, this general topic has engendered much interest but little published research to date.

In this context, the focus of our work is on exploring the performance and scalability of a common visualization algorithm – raycasting volume rendering – implemented with different parallel programming models and run on a large supercomputer comprised of six-core chips. In this study, we compare a traditional implementation based on message-passing against a "hybrid" parallel implementation, which uses a blend of traditional message-passing (inter-chip) and shared-memory (intra-chip) parallelism. The thesis we wish to test is whether there are opportunities in the hybrid-parallel implementation for performance and scalability gains that result from using shared-memory parallelism among cores within a chip.

Over the years, there has been a consistent and well-documented concern that the overall runtime of large-data visualization algorithms is dominated by I/O costs (e.g., [2], [3], [4]). During our experiments, we observed results consistent with previous work: there is a significant cost associated with scientific data I/O. In this study, however, we focus exclusively on the performance and scalability of the ray casting volume rendering algorithm, not on parallel I/O performance. This approach is valid for many visualization use cases, such as creating multiple images from a single dataset that fits entirely within the memory footprint of a large system, or creating one or more images of data that is already resident in memory, as in the case of *in-situ* visualization.

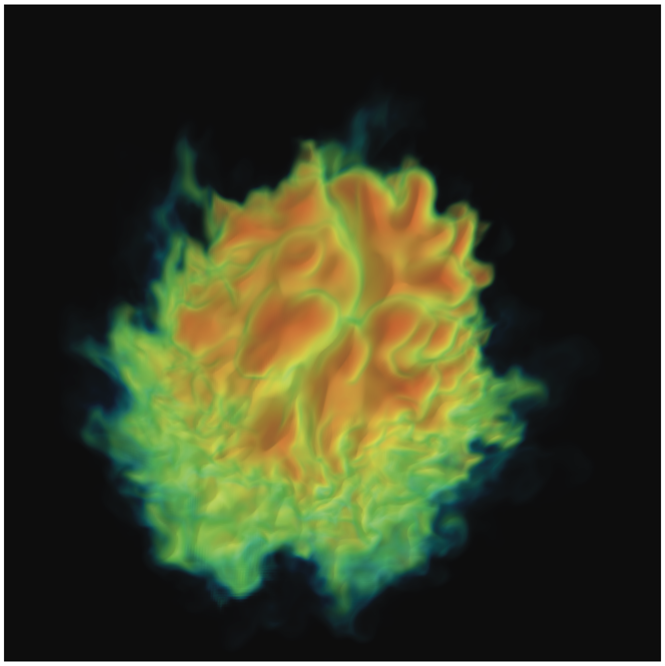


Fig. 1. This 4608^2 image of a combustion simulation result was rendered by our MPI+pthreads implementation running on 216,000 cores of the JaguarPF supercomputer.

Our findings (Section IV) show that there is indeed opportunity for performance gains when using hybrid-parallelism for raycasting volume rendering across a wide range of concurrency levels. The hybrid-parallel implementation runs faster, requires less memory, and for this particular algorithm and set of implementation choices (Section III), consumes less communication bandwidth than the traditional, MPI-only implementation.

II. Background and Previous Work

A. Parallel Volume Rendering

Volume rendering is a common technique for displaying 2D projections of 3D sampled data [5], [6] and is computationally, memory, and data I/O intensive. In the quest towards interactivity, as well as to address the challenges posed by growing data size and complexity, there has been a great deal of work over the years in the space of parallel volume visualization (see Kaufman and Mueller [7] for an overview).

Within the field of parallel volume rendering, our work focuses on examining a form of "hybrid parallelism," which is distinct and different from the term "hybrid volume rendering." Hybrid volume rendering refers to using combination of image- and object-order techniques to perform rendering [7]. In contrast, hybrid parallel refers to the use of a mixture of shared- and distributed-memory algorithms, programming, and execution environments [8].

However, our hybrid-parallel implementation also makes use of hybrid volume rendering (e.g. [9], [10], [11], [12]), starting with an object-order partitioning to distribute source data blocks to processors where they are rendered using ray casting [5], [13], [6], [14]. Within a processor, we then use an image-space decomposition, similar to [15], to allow multiple rendering threads to cooperatively generate partial images that are later combined via compositing into a final image ([6], [5], [14]).

Hybrid volume rendering techniques have proven successful in achieving scalability and tackling large data sizes. The TREX system [3] is a hybrid volume rendering algorithm on a shared-memory platform that uses object-parallel data domain decomposition and texture-based, hardware-accelerated rendering followed by a parallel, software-based composition phase with image-space partitioning. The design choices for which part of the SGI Origin to use for different portions of the algorithm reflect a desire to achieve optimal performance at each algorithmic stage and to minimize inter-stage communication costs. Childs et al. [16] present a hybrid scheme for volume rendering massive datasets (with one hundred million unstructured elements and a 3000^3 rectilinear data set). Their approach parallelizes over both input data elements and output pixels, and is demonstrated to scale well on up to 400 processors. Peterka et al. were the first to run a hybrid volume rendering algorithm at massive concurrency, rendering 4480^3 data sizes with 32,000 cores on an IBM BG/P system [4]. They demonstrated generally good scalability and found that the compositing phase slowed down when more than ten thousand cores were involved, likely due to hardware or MPI limitations. To address this problem, they reduced the number of processors involved in the compositing phase.

The most substantial difference between our work and previous work in hybrid volume rendering is that we are exploiting hybrid parallelism. In addition, we contribute to the knowledge of hybrid volume rendering at massive concurrency: we performed experiments with up to 216,000 cores, which is more than six times larger than previously published results. Further, our study reproduces the degradation in compositing performance first discovered by Peterka et al. [4] via a different software implementation and supercomputer, and we present results that characterize this effect in more detail.

B. Traditional and Hybrid Parallelism

The Message Passing Interface (MPI) evolved as the defacto standard for parallel programming and execution on machines consisting of single-core CPUs interconnected via a high-speed fabric [17]. To use MPI, an application developer must explicitly add MPI library calls to an application to implement fundamental parallel execution motifs: data scatter and gather, execution synchronization, and so forth. In MPI parlance, a *processing element* (PE) is the fundamental unit of execution, and historically each MPI PE has mapped one-to-one to the processors of a massively parallel (MPP) system. To support more recent multi-core MPPs, vendors' MPI implementations provide support for MPI PEs to be mapped onto one or more cores of multi-core chips. However, on multi-core platforms, there may be opportunities for more efficient inter-PE communication through local, high-speed, shared memory that bypasses the MPI interface.

Shared-memory parallel applications are somewhat easier to develop than distributed memory ones as there is no need to explicitly move data among parallel program elements. Instead, each *execution thread* has access to the same shared memory within a single address space. Common programming models for shared-memory parallelism include POSIX threads [18], OpenMP [19], and Intel Thread Building Blocks [20]. These APIs allow applications to manage creation and termination of execution threads, and synchronize thread execution through semaphores and mutexes. The scalability of shared-memory codes is typically limited by physical constraints: there are typically only a few cores in a single CPU. Four to six cores per CPU are common today, although the trend seems to be towards hundreds to thousands of cores per chip.

In all of the above models, the developer must explicitly design for parallelism, as opposed to relying on a compiler to discover and implement parallelism. Other approaches, which allow the developer to express parallelism implicitly via language syntax, include data-parallel languages (e.g., CUDA [21]), languages with data parallel extensions (e.g., High Performance Fortran [22]), and partitioned global address space (PGAS) languages (e.g., Unified Parallel C (UPC) [23], which offers the developer a single-address space view of memory, even on distributed memory platforms).

Hybrid parallelism has evolved in response to the widespread deployment of multi-core chips in distributed-memory systems. An MPI-hybrid model allows data movement among nodes using traditional MPI motifs like scatter and gather, but within nodes using shared-memory parallelism via POSIX threads or OpenMP. Previous work comparing MPI-only with MPI-hybrid implementations (e.g., [24], [25]) has focused on benchmarking well-known computational kernels. In contrast, our study examines this space from the perspective of visualization algorithms.

The previous studies point to several areas where MPI-hybrid may outperform MPI-only. First, MPI-hybrid tends to require less memory for applications with domain decomposition (e.g. parallel volume rendering), since fewer domains means less "surface area" between domains and hence less exchange of "ghost zone" data. Second, the MPI runtime allocates various tables, buffers, and constants on a per-PE basis. Today, the gain from using fewer PEs to reduce this memory overhead may seem small with only four or six cores per chip, but the trend towards hundreds of cores per chip with less memory per core will magnify these gains. Third, MPI-hybrid can use only one PE per node for collective operations such as scatter, gather and all-to-all, thereby reducing the absolute number of messages traversing the inter-connect. While the size of the messages in this scenario may be larger or smaller depending upon the specific problem, a significant factor influencing overall communication performance is latency, which is reduced by using fewer messages.

The primary contributions of our work is a comparison of the performance and resource requirements of hybrid- and traditional-parallel implementations of raycasting volume rendering. Our methodology and results (Section IV) are consistent with those identified in previous studies of hybrid parallelism from the HPC community (e.g., [24]).

III. Implementation

From a high level view, our parallel volume rendering implementation repeats a design pattern that forms the basis of previous work (e.g., [9], [10], [11], [12]). Given a source data volume S and n parallel processes, each process reads in $\frac{1}{n}$ of S. Next, each of the n processes performs raycasting volume rendering on its data subdomain to produce a set of image fragments. Next, each of the n processes participates in a compositing stage in which fragments are exchanged and combined into a final image. Each of the n processes then sends it portion of the completed image to process 0 for display or I/O to storage. Figure 2 provides a block-level view of this organization. Our distributed memory parallel implementation is written in C++ and C and makes calls to MPI [17]. The portions of the implementation that are shared-memory parallel are written using a combination of C++ and C and either POSIX threads [18] or OpenMP [19], so that we are actually comparing two hybrid implementations that we refer to as MPI+pthreads and MPI+OpenMP.

The MPI-only and MPI-hybrid implementations differ in several key respects. First, the raycasting volume rendering algorithm itself is serial on each MPI-only PE or is shared-memory multicore-parallel in the MPI-hybrid case. We discuss this issue in more detail in Section III-A. Second, the communication topology in the compositing stage differs slightly, and we discuss this issue in more detail in Section III-B. A third difference is in how data is partitioned across the pool of parallel processes. In the MPI-only implementation, each PE loads and operates on a disjoint block of data. In the MPI-hybrid case, each MPI PE loads a disjoint block of data and each of its worker threads operate in parallel on that data using an image-parallel decomposition [15].

A. *Parallel, Multicore Raycasting Volume Rendering*

Our raycasting volume rendering code implements Levoy's method [5]: we compute the intersection of a ray with the data block, and then compute the color at a fixed step size along the ray through the volume. All such colors along the ray are integrated front-to-back using the "over" operator. Output from our algorithm consists of a set of image fragments that contain an x, y pixel location, R, G, B, α color information, and a z-coordinate. The z-coordinate is the location in eye coordinates where the ray penetrates the block of data. Later, these fragments are composited in the correct order to produce a final image (see Section III-B).

In the MPI-only case, this serial implementation is invoked on each of the MPI PEs. Each operates on its own disjoint block of data. As we are processing structured rectilinear grids, all data subdomains are spatially disjoint, so we can safely use the ray's entry point into the data block as the z-coordinate for sorting during the later composition step.

In the MPI-hybrid case, the raycasting volume renderer on each MPI PE is a shared-memory parallel implementation

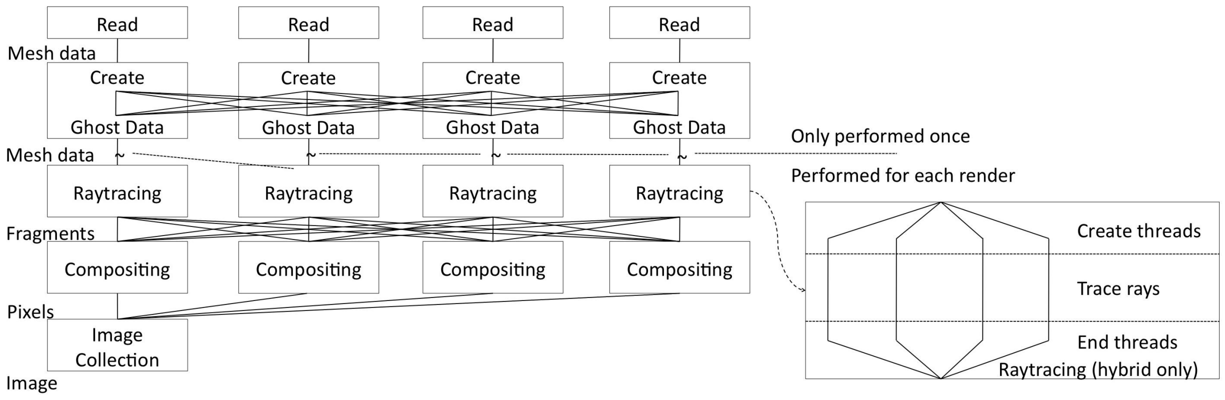


Fig. 2. Diagram of our system architecture.

consisting of T threads all executing concurrently to perform the raycasting on a single block of data. As in [15], we use an image-space partitioning: each thread is responsible for raycasting portion of the image. Our image-space partitioning is interleaved, where the image is divided into many small tiles that are distributed amongst the threads. Using a process similar to autotuning [26], we determined the set of algorithmic parameters and choices that produce consistently good performance on a variety of source data sizes on the six-core AMD Opteron processor (Section IV-B). The set of optimal tunable algorithmic parameters and choices consists of an image tile size of 32×32 pixels combined with dynamic work assignment of tiles to threads to minimize load imbalance.

B. Parallel Compositing

The compositing algorithm takes the fragments generated by the raycasting algorithm and produces the final image. The compositing begins by partitioning the pixels of the final image amongst the MPI PEs. It then performs an all-to-all communication step, where MPI PEs send and receive fragments. The sent fragments are the fragments from that MPI PE's raycasting phase. The MPI PE identifies where to send fragments using the partition information. The received fragments correspond to the pixels of the final image that that MPI PE owns. The fragments are exchanged using an `MPI_Alltoallv` call. This call provides the same functionality as direct sends and receives, but bundles the messages into fewer point-to-point exchanges, and thus is more efficient. After the fragments are exchanged, each MPI PE is able to perform the final compositing for its portion of the larger image, using the "over" operator on the fragments. The final step is to collect the image subpieces to MPI PE 0 and output the entire image.

Peterka et al. [4] reported scaling difficulties for compositing when using more than 8,000 MPI PEs. They solved this problem by reducing the number of MPI PEs receiving fragments to be no more than 2,000. We emulated this approach, again limiting the number of MPI PEs receiving fragments, although we experimented with values higher than 2,000.

In the hybrid implementations, only one thread per socket participates in the compositing phase. That thread gathers fragments from all other threads in the same socket, packs them into a single buffer, and transmits them to other compositing PEs. This approach results in fewer messages than if all threads in the hybrid parallel implementation were to send messages to all other threads on all other CPUs. Our aim here is to better understand the opportunities for improving performance in the hybrid-parallel implementation. The overall effect of this design choice is an improvement in communication characteristics, as indicated in Section IV-E2.

IV. Results

A. Methodology

Our methodology is designed to test the hypothesis that an MPI-hybrid implementation exhibits better performance and resource utilization than the MPI-only implementation. We compare the cost of MPI runtime overhead and corresponding memory footprint in Section IV-D1; compare the absolute amount of memory required for data blocks and ghost zone (halo) exchange in Section IV-D2; compare scalability of the raycasting and compositing algorithms in Sections IV-E1 and IV-E2; and compare the levels of communication required during the compositing phase in Section IV-E2.

Our test system, JaguarPF, is a Cray XT5 located at Oak Ridge National Lab and was recently ranked by the Top500 list as the fastest supercomputer in the world with a theoretical peak performance of 2.3 Petaflops [27]. Each of the 18,688 nodes has two sockets, and each socket has a six-core 2.6GHz AMD Opteron processor, for a total of 224,256 compute cores. With 16GB per node (8GB per socket), the system has 292TB of aggregate memory and roughly 1.3GB per core.

We conducted the following three studies:

- **strong scaling**, in which we fixed the image size at 4608^2 and the dataset size at 4608^3 (97.8 billion cells) for all concurrency levels;
- **weak–dataset scaling**, with the same fixed 4608^2 image, but a dataset size increasing with concurrency up to 23040^3 (12.2 trillion cells) at 216,000-way parallel; and
- **weak scaling**, in which we increased both the image and dataset up to 23040^2 and 23040^3, respectfully, at 216,000-way parallel.

Note that at the lowest concurrency level, all three cases coincide with a 4608^2 image and 4608^3 dataset size.

In the hybrid case, we shared a data block among six threads and used one sixth as many MPI PEs. Although we could have shared a data block among as many as twelve threads on each dual-socket six-core node, we chose not to because sharing data across sockets results in non-uniform memory access. Based on preliminary tests, we estimated this penalty to be around 5 or 10% of the raycasting time. Therefore, we used six threads running on the cores of a single six-core processor.

Because the time to render is view-dependent, we executed each raycasting phase ten times over a selection of ten different camera locations (see Figure 3). The raycasting times we report are an average over all locations.

In the compositing phase, we tested either four or five (depending on memory constraints) ratios of total PEs to compositing PEs. We restricted the compositing experiment to only two views (the first and last) because it was too costly to run a complete battery of all view and ratio permutations. Since the runtime of each trial can vary due to contention for JaguarPF's interconnection fabric with other users, we ran the compositing phase ten times for both views. We report mean and minimum times over this set of twenty trials. Minimum times most accurately represent what the system is capable of under optimal, contention-free conditions, while mean times help characterize the variability of the trials.

B. Determining Optimal Algorithmic Parameters

In an image-space partitioning for shared-memory parallel volume rendering, one tunable algorithmic parameter is the size of the image tile resulting from partitioning the final image space. Our objective here is to determine whether or not some image tile sizes and shapes result in better performance than others, and if so, to select an image tile size that results in better overall performance for this particular problem on the six-core AMD Opteron processor.

The approach we use is to run the shared-memory parallel volume rendering code on a 384^3 mesh on a single socket at six-way concurrency and vary the size of the image tile size over a range from $w = [2^1, 2^2, \ldots, 2^8, 2^9]$ and $h = [2^1, 2^2, \ldots, 2^8, 2^9]$ for a total of 100 different tile sizes and shapes. For each tile size/shape, we measure (frame rendering time) FRT over thirty different views. Also, we are using dynamic rather than static work assignments as we saw earlier that the dynamic assignments results in better overall load balance for this particular problem configuration. For this problem, there are three "invalid" block size configurations, 256×512, 512×512, and 512×256, that produce decompositions resulting in only one or two total work blocks.

From the test results, shown in Figure 4, we see clear "sweet" and "sour" spots in performance. To produce that figure, we "normalized" FRT by the minimum so that values range from $[1.0 \ldots \infty]$. In this particular set of results, the maximum data value is about 3.0, which means the FRT of the worst-performing tile size was about three times as slow as the best-performing configuration. We see that very small and very large image tiles produce poorer performance: at small tile sizes, the threads' access to the work list is serialized through a mutex and performance is adversely affected. At large tile sizes, there are not enough tiles to result in good load balance. Therefore, the sweet spot tends to fall in regions of medium-sized tiles.

An interesting feature in Figure 4 is the relatively poor performance for block widths $w = [2^0, 2^1, 2^2]$. At these configurations, we observe in data collected using PAPI a relatively high level of L1, L2, and TLB cache misses (these results are not shown due to space limitations). While the reason for the high cache misses at these tile sizes is not clear, there is an obvious correlation between cache miss rates and FRT. Also, the main point here is that our approach, which is consistent with autotuning in general, is to emperically performance across a range of algorithmic parameters to determine the configurations that perform better and worse for a given problem configuration on a given platform. The alternative is to attempt to derive a quantitative performance model of a complex system.

Based upon the results of this study, we decided to perform our scalability studies using an image tile size of 32×32 pixels. That tile size lies within the sweet spot that is visible in Figure 4. Other tile size choices could be equally valid: we

Color	Camera
Black	$(1, 0, 0)$
Black	$(0, 1, 0)$
Black	$(0, 0, 1)$
Cyan	$(0, 1, 1)$
Magenta	$(1, 0, 1)$
Yellow	$(1, 1, 0)$
Red	$(2, 1, 1)$
Green	$(1, 2, 1)$
Blue	$(1, 1, 2)$
Gray	$(1, 1, 1)$

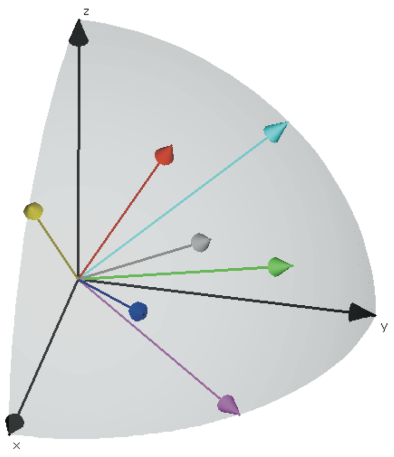


Fig. 3. The raycasting phase used these ten camera locations to approximate an "average" use case.

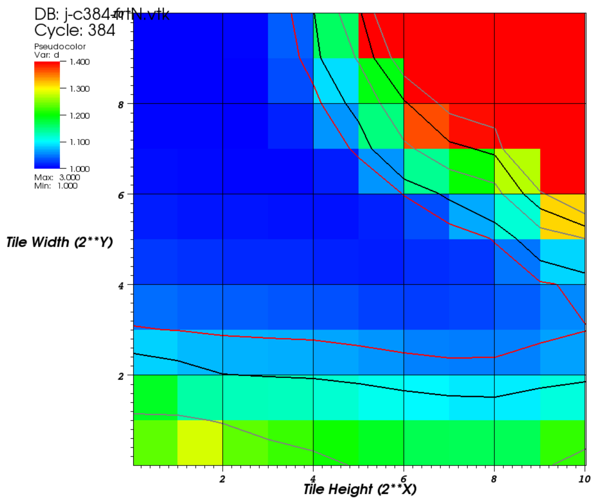


Fig. 4. Normalized frame rate timings resulting from different sized image tiles. Poorer performance results when using very large or very small work blocks, better performance results when using medium-sized blocks.

			Strong Scaling			*Weak Scaling*
Concurrency	MPI-only PEs	MPI-hybrid PEs	MPI-only Block	MPI-hybrid Block	Memory per Node	Dataset Size
12^3	1728	288	$384 \times 384 \times 384$	$384 \times 768 \times 1152$	10368MB	4608^3
24^3	13824	2304	$192 \times 192 \times 192$	$192 \times 384 \times 576$	1296MB	9216^3
36^3	46656	7776	$128 \times 128 \times 128$	$128 \times 256 \times 384$	384MB	13824^3
48^3	110592	18432	$96 \times 96 \times 96$	$96 \times 192 \times 288$	162MB	18432^3
60^3	216000	36000	$76 \times 76 \times 76$	$76 \times 153 \times 230$	80.4MB / 81.6MB	23040^3

TABLE I
PROBLEM SIZE CONFIGURATIONS AND PER-NODE MEMORY REQUIREMENTS FOR OUR EXPERIMENT. THE MEMORY USAGE FOR MPI-HYBRID AND MPI-ONLY AT 216,000 CORES DIFFERS BECAUSE THEY HAVE DIFFERENT DATASET DIMENSIONS.

Cores	Mode	MPI PEs	MPI Runtime Memory Usage		
			Per PE (MB)	Per Node (MB)	Aggregate (GB)
1728	MPI-hybrid	288	67	133	19
1728	MPI-only	1728	67	807	113
13824	MPI-hybrid	2304	67	134	151
13824	MPI-only	13824	71	857	965
46656	MPI-hybrid	7776	68	136	518
46656	MPI-only	46656	88	1055	4007
110592	MPI-hybrid	18432	73	146	1318
110592	MPI-only	110592	121	1453	13078
216000	MPI-hybrid	36000	82	165	2892
216000	MPI-only	216000	176	2106	37023

TABLE II
MEMORY USAGE AS MEASURED DIRECTLY AFTER MPI INITIALIZATION.

see that tile sizes having relatively large width and relatively small height also lie within the sweet spot of FRT.

We ran a similar study aimed at exploring the impact on FRT when the algorithm uses either static or dynamic work assignments. The results of that study, not shown here due to space limitations, indicates that the static assignment approach produces slightly better FRT values at the small block sizes. The dynamic approach produces better results at medium and larger block sizes. Therefore, for our weak- and strong-scaling runs, we chose to use the dynamic work assignment since it results in better performance than static assignments at 32×32 pixel tile sizes.

C. Source Data and Decomposition

Starting with a 512^3 dataset of combustion simulation results [1], we used trilinear interpolation to upscale it to arbitrary sizes in memory. We scaled equally in all three dimensions to maintain a cubic volume. Our goal was to choose a problem size that came close to filling all available memory (see Table I). Although upscaling may distort the results for a data-dependent algorithm, the only data dependency during raycasting is early ray termination. However, we found that for our particular dataset and transfer function, there was always at least one data block for which no early terminations occurred. Moreover, the cost of the extra conditional statement inside the ray integration loop to test for early termination added a 5% overhead. Therefore, we ran our study with early ray termination turned off, and we believe that upscaling the dataset does not effect our results.

Because the compute nodes on an XT5 system have no physical disk for swap, and hence no virtual memory, exceeding the physical amount of memory causes program termination. Our total memory footprint was four times the number of bytes in the entire dataset: one for the dataset itself, and the other three for the gradient data, which we computed by central difference and used in shading calculations. Although each node has 16GB of memory, we could reliably allocate only 10.4GB for the data block and gradient field at 1,728-way concurrency because of overhead from the operating system and MPI runtime.

[1]Sample data courtesy J. Bell and M. Day, Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory.

We chose concurrencies that are cubic numbers to allow for a clean decomposition of the entire volume into cubic blocks per MPI PE. In the MPI-hybrid case, however, these blocks are rectangular because we aggregated six blocks $(1 \times 2 \times 3)$ into one shared block.

1) Strong Scaling: We used a fixed 4608^3 dataset at all concurrencies, except for 216,000-way where the 4608^3 dataset could not be evenly divided. With increasing concurrency, each MPI PE was assigned a correspondingly smaller data block, as seen in Table I.

At 216,000-way concurrency we rounded down to a 4560^3 dataset for MPI-only and a $4560 \times 4590 \times 4600$ dataset for MPI-hybrid. As a result, the MPI-only dataset is approximately 1.4% smaller than the MPI-hybrid dataset. While this difference might seem to give an advantage to the MPI-only implementation, results in later sections show that the MPI-hybrid performance and resource utilization are uniformly better than for MPI-only.

2) Weak Scaling: For MPI-only, we used a 384^3 block for each MPI PE at all concurrency levels. For MPI-hybrid, we aggregated this into a $384 \times 786 \times 1152$ block. As a result, we processed datasets with sizes ranging from 4608^3 to 23040^3 (see Table I).

D. Memory Usage

1) Overhead at Initialization: Because MPI-hybrid uses fewer MPI PEs, it incurs less memory overhead from the MPI runtime environment and from program-specific data structures that are allocated per PE. Table II shows the memory footprint of the program as measured directly after calling `MPI_Init` and reading in command-line parameters. We collected the `VmRSS`, or "resident set size," value from the `/proc/self/status` interface. Memory usage was sampled only from MPI PEs 0 through 6, but those values agreed within 1–2%. Therefore, the per-PE values we report in Table II are from PE 0 and the per-node and aggregate values are derived from the per-task value: MPI-only uses twelve PEs per node versus MPI-hybrid's two PEs per node, and in the aggregate MPI-hybrid has one sixth as many PEs. At 216,000 cores, the per-PE runtime overhead of MPI-only is more that $2\times$ that of MPI-hybrid and the per-node and aggregate memory usage is another factor of six larger for

MPI-only because it uses 6× as many PEs as MPI-hybrid. Thus, MPI-only uses nearly 12× as much memory per-node and in-aggregate than MPI-hybrid to initialize the MPI runtime at 216,000-way concurrency.

2) Ghost Data: Two layers of ghost data are required in our raycasting phase: the first layer for trilinear interpolation of sampled values, and the second layer for computing the gradient field using central differences (gradients are not pre-computed for our dataset). Because the MPI-hybrid approach uses fewer, larger blocks in its decomposition, it requires less exchange and storage of ghost data by roughly 40% across all concurrency levels and for both strong and weak scaling (see Figure 5).

E. Scaling Study

1) Raycasting: Our raycasting phase scales nearly linearly because it involves no inter-processor communication (see Figure 6). Each MPI PE obtains its data block, then launches either one (MPI-only) or six (MPI-hybrid) raycasting threads. Working independently, each thread tests for ray-triangle intersections along the data block's bounding box and, in the case of a hit, integrates the ray by sampling data values at a fixed interval along the ray, applying a transfer function to the values, and aggregating the resulting color and opacity in a fragment for that ray's image position. For these runs and timings, we use trilinear interpolation for data sampling along the ray as well as a Phong-style shader. The final raycasting time is essentially the runtime of the thread that takes the most integration steps. This behavior is entirely dependent on the view. Our approach, which is aimed at understanding "average" behavior, uses ten different views (see Figure 3 in Section IV-A) and reports their average runtime.

Overall, we have achieved linear scaling up to 216,000 for the raycasting phase with MPI-only (see Figure 6). MPI-hybrid exhibits different scaling behavior because it has a different decomposition geometry: MPI-only has a perfectly cubic decomposition, while MPI-hybrid aggregates $1 \times 2 \times 3$ cubic blocks into rectangular blocks that are longest in the z-direction (see Table I). The interaction of the decomposition geometry and the camera direction determine the maximum number of ray integration steps, which is the limiting factor for the raycasting time. At lower concurrencies, this interaction benefits MPI-hybrid, which outperforms MPI-only by as much as 11% (see Table III). At higher concurrencies the trend flips, and MPI-only outperforms MPI-hybrid by 10%. We expect that if we were able to run on an eight-core system with a $2 \times 2 \times 2$ aggregation factor for MPI-hybrid, both implementations would scale identically. We also note that at 216,000 cores, raycasting is less than 20% of the total runtime (see Figure 7), and MPI-hybrid is over 50% faster because of gains in the compositing phase that we describe next.

2) Compositing: We observe the same effect that Peterka et al. [4] report: for higher concurrencies, it is more beneficial to use only a subset of PEs for compositing. While Peterka et al. found that 1,000 to 2,000 compositors were optimal for up to 32,768 total PEs, we have found that at 46,656 PEs and above the optimal number of compositors is closer to 4,000 to 8,000 (see Figure 8). We note that there are many differences between our study and theirs in the levels of concurrency, architectures, operating systems, communication networks, and MPI libraries, each potentially introducing variation in the ideal number of compositors.

Above 1,728 cores, we observe that the compositing times are systematically better for the MPI-hybrid implementation. The primary cost of compositing is the `MPI_Alltoallv` call that moves each fragment from the PE where it originated during raycasting to the compositing PE that owns the region of image space where the fragment lies. Because MPI-hybrid aggregates these fragments in the memory shared by six threads, it uses on average about 6× fewer messages than MPI-only (see Figure 9). In addition, MPI-hybrid exchanges less fragment data because its larger data blocks allow for more compositing to take place during ray integration.

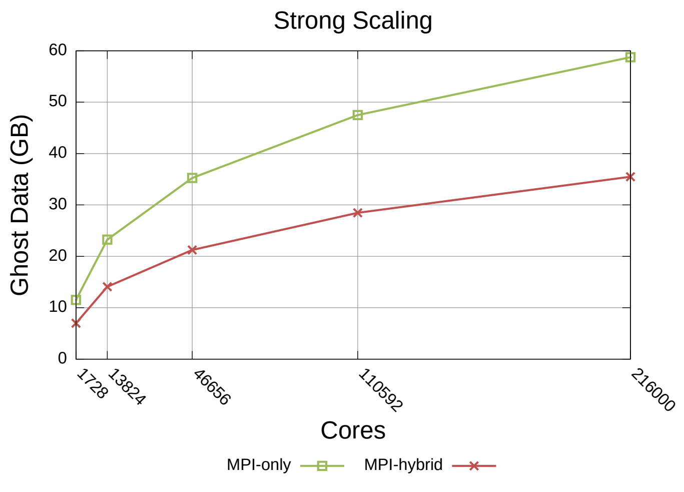
We observe a critical point in the compositing performance beginning at 13,824 cores where the elapsed time for MPI-only begins to increase with additional compositors. A similar critical point occurs for MPI-hybrid at 46,656 cores. We believe this critical point exists due to the characteristics of the underlying interconnect fabric. Because MPI-hybrid generates fewer and smaller messages, the critical point occurs at a higher level of concurrency than for MPI-only. Also, our MPI-hybrid version outperforms the MPI-only version across all configurations in these tests. Future work will examine this issue in more detail and on different architectures to better illuminate factors contributing to performance of this stage of processing.

3) Overall Performance: At 216,000 cores, the best compositing time for MPI-hybrid (0.35s, 4500 compositors) is 67% less than for MPI-only (1.06s, 6750 compositors). Furthermore, at this scale compositing time dominates rendering time, which is roughly 0.2s for both MPI-only and MPI-hybrid. Thus, the total render time is 55% faster for MPI-hybrid (0.56s versus 1.25s). Overall, the scaling study shows that the advantage of MPI-hybrid over MPI-only becomes greater as the number of cores increases (see Figure 7).

V. Conclusion and Future Work

The multi-core era offers new opportunities and challenges for parallel applications. This study has shown that hybrid parallelism offers performance improvements and better resource utilization for raycasting volume rendering on a large, distributed-memory supercomputer comprised of multi-core CPUs. The advantages we observe for hybrid parallelism are reduced memory footprint, reduced MPI overhead, and reduced communication traffic. These advantages are likely to become more pronounced in the future as the number of cores per CPU increases while per-core memory size and bandwidth decrease.

We found that at high concurrency, fragment exchange during the compositing phase is the most expensive operation. Our compositing algorithm relies entirely on the implementation of the `MPI_Alltoallv` call in the Cray MPI library.



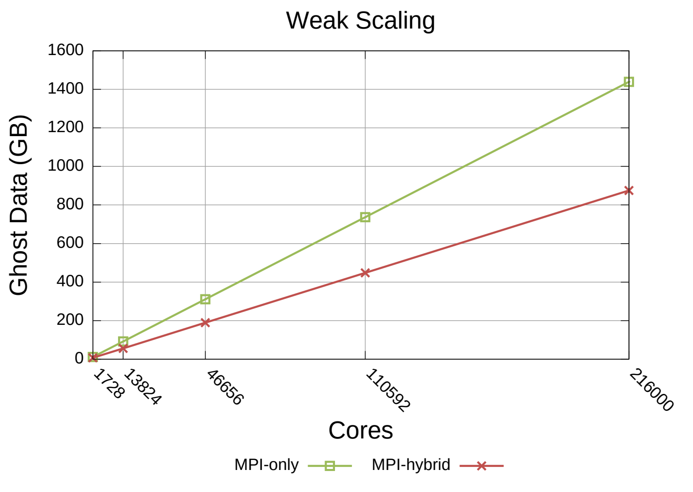
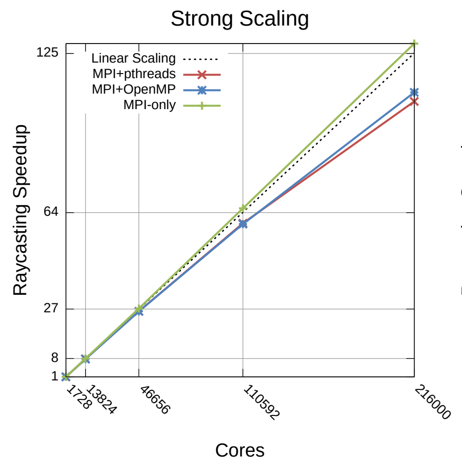


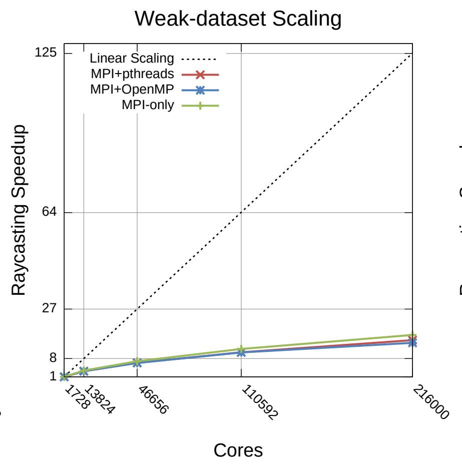
Fig. 5. Ghost data required by our implementations.

	Strong Scaling			*Weak–dataset Scaling*			*Weak Scaling*		
Cores	MPI-only	MPI+pthreads	MPI+OpenMP	MPI-only	MPI+pthreads	MPI+OpenMP	MPI-only	MPI+pthreads	MPI+OpenMP
1728	24.88	22.31	22.23	24.88	22.31	22.23	24.88	22.31	22.23
13824	3.10	2.84	2.83	7.33	7.06	7.06	26.65	24.92	24.93
46656	0.92	0.85	0.85	3.56	3.51	3.51	27.18	25.88	25.88
110592	0.38	0.37	0.37	2.13	2.15	2.15	27.45	26.59	26.50
216000	0.19	0.21	0.20	1.46	1.48	1.58	22.17	26.79	26.88

TABLE III
RAYCASTING TIMES FOR OUR THREE IMPLEMENTATIONS.

Our finding that using a subset of compositing PEs reduces communication overhead agrees with what Peterka et al. [4] found for the MPI implementation on an IBM BG/P system, suggesting that both libraries use similar implementations and optimizations of `MPI_Alltoallv`. A separate paper by Peterka et al. [28] introduces a new compositing algorithm, "Radix-k," that shows good scaling up to 16,000-way concurrency. This approach, which is compatible with our hybrid parallel system, may lead to even faster compositing times. Studying their combined performance, especially at concurrencies in the hundreds of thousands, is an avenue for future work.





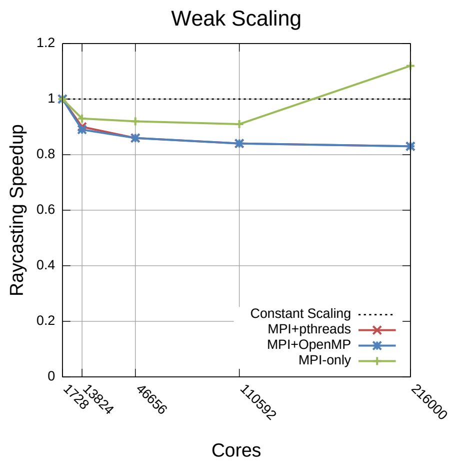


Fig. 6. The speedups (referenced to 1,728 cores) for both the raycasting phase and the total render time (raycasting and compositing). The raycasting speedup is linear for MPI-only, but sublinear for MPI-hybrid: this effect is caused by the difference in decomposition geometries (cubic versus rectangular).

Acknowledgment

This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET). This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Additionally, preliminary results for this work were collected using resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] C. T. Silva, A. E. Kaufman, and C. Pavlakos, "PVR: High-Performance Volume Rendering," *Computing in Science and Engineering*, vol. 3, no. 4, pp. 18–28, 1996.

[3] J. Kniss, P. McCormick, A. McPherson, J. Ahrens, J. Painter, A. Keahey, and C. Hansen, "Interactive Texture-Based Volume Rendering for Large Data Sets," *IEEE Computer Graphics and Applications*, July/August 2001.

[4] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham, "End-to-end study of parallel volume rendering on the ibm blue gene/p," in *Proceedings of ICPP'09 Conference*, September 2009.

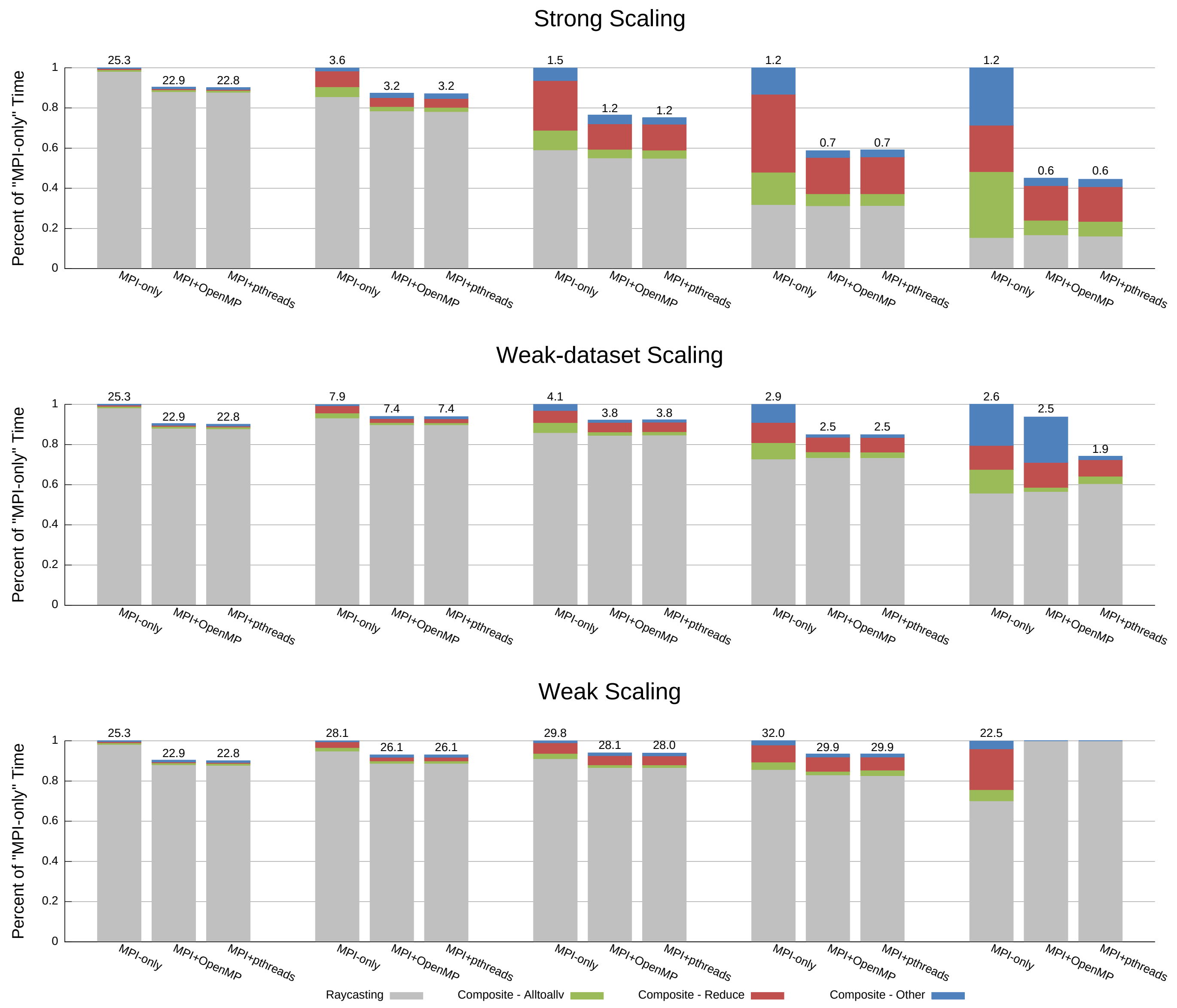


Fig. 7. In terms of total render time, MPI-hybrid outperforms MPI-only at every concurrency level, with performance gains improving at higher concurrency.

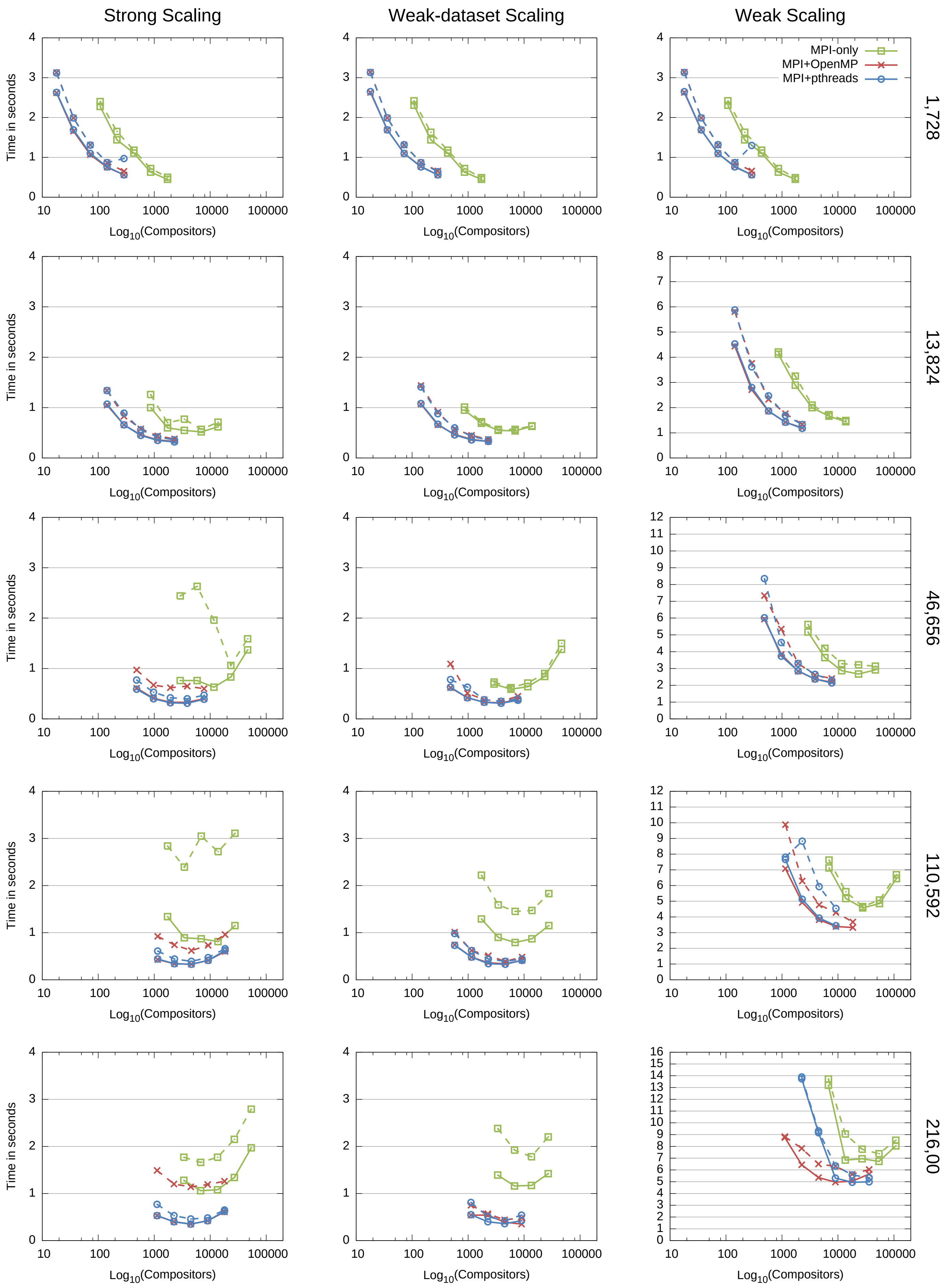


Fig. 8. Compositing times for different ratios of compositing PEs to total PEs. Solid lines show minimum times taken over ten trials each for two different views; dashed lines show the corresponding mean times.

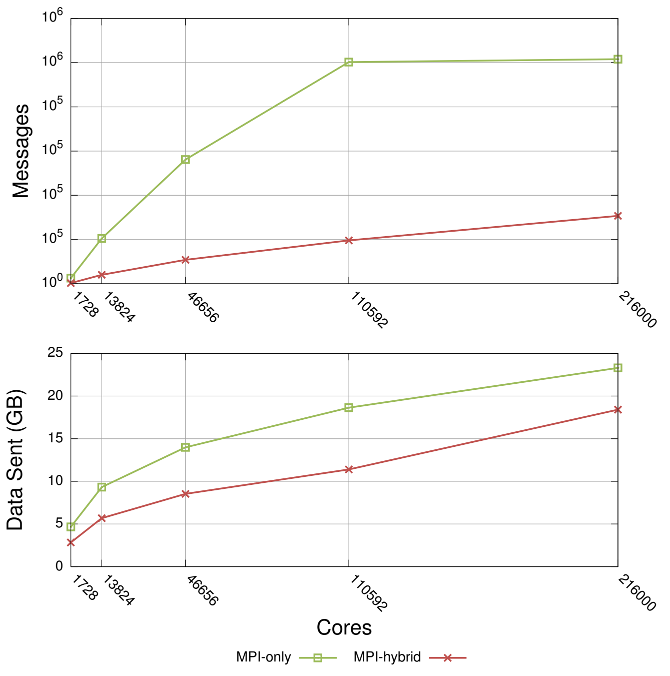


Fig. 9. The number of messages and total data sent during the fragment exchange in the compositing phase.

[5] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, May 1988.

[6] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 65–74, 1988.

[7] A. Kaufman and K. Mueller, "Overview of Volume Rendering," in *The Visualization Handbook*, C. D. Hansen and C. R. Johnson, Eds. Elsevier, 2005, pp. 127–174.

[8] L. M. Liebrock and S. P. Goudy, "Methodology for Modelling SPMD Hybrid Parallel Computation," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 8, pp. 903–940, 2008.

[9] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," in *Proceedings of the 1993 Parallel Rendering Symposium*. ACM Press, October 1993, pp. 15–22.

[10] R. Tiwari and T. L. Huntsberger, "A Distributed Memory Algorithm for Volume Rendering," in *Scalable High Performance Computing Conference*, Knoxville, TN, USA, May 1994.

[11] K.-L. Ma, "Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures," in *PRS '95: Proceedings of the IEEE symposium on Parallel rendering*. New York, NY, USA: ACM, 1995, pp. 23–30.

[12] C. Bajaj, I. Ihm, G. Joo, and S. Park, "Parallel ray casting of visibly human on distributed memory architectures," in *VisSym'99 Joint EUROGRAPHICS-IEEE TVCG Symposium on Visualization*, 1999, pp. 269–276.

[13] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 51–58, 1988.

[14] C. Upson and M. Keeler, "V-buffer: visible volume rendering," in *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1988, pp. 59–64.

[15] J. Nieh and M. Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures," in *Proceedings of the 1992 Workshop on Volume Visualization*. ACM SIGGRAPH, October 1992, pp. 17–24.

[16] H. Childs, M. A. Duchaineau, and K.-L. Ma, "A scalable, hybrid scheme for volume rendering massive data sets," in *Eurographics Symposium on Parallel Graphics and Visualization*, 2006, pp. 153–162.

[17] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI – The Complete Reference: The MPI Core, 2nd edition*. Cambridge, MA, USA: MIT Press, 1998.

[18] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[19] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[20] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor*. O'Reilly Media Inc., 2007.

[21] NVIDIA Corporation, *NVIDIA CUDA^{TM} Version 2.1 Programming Guide*, 2008.

[22] High Performance Fortran Forum, "High Performance Fortran language specification, version 1.0," Houston, Tex., Tech. Rep. CRPC-TR92225, 1993. [Online]. Available: citeseer.ist.psu.edu/fortran92high.html

[23] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC – Distributed Shared Memory Programming*. John Wiley & Sons, 2005.

[24] G. Hager, G. Jost, and R. Rabenseifner, "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes," in *Proceedings of Cray User Group Conference*, 2009.

[25] D. Mallón, G. Taboada, C. Teijeiro, J. Tourino, B. Fraguela, A. Gómez, R. Doallo, and J. Mourino, "Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures," in *16th European PVM/MPI Users' Group Meeting, (EuroPVM/MPI'09)*, September 2009.

[26] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[27] "The top 500 supercomputers," 2009, http://www.top500.org.

[28] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur, "A configurable algorithm for parallel image-compositing applications," in *Supercomputing '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 1–10.